
Mohammadhosein Hasanbeig, Natasha Yogananda Jeppu, Alessandro Abate, Tom Melham, Daniel Kroening*

Computer Science Department, University of Oxford, Parks Road, Oxford, United Kingdom, OX1 3QD
{hosein.hasanbeig, natasha.yogananda.jeppu, alessandro.abate, tom.melham} @cs.ox.ac.uk and daniel.kroening@magd.ox.ac.uk

Abstract

This paper proposes DeepSynth, a method for effective training of deep Reinforcement Learning (RL) agents when the reward is sparse and non-Markovian, but at the same time progress towards the reward requires achieving an unknown sequence of high-level objectives. Our method employs a novel algorithm for synthesis of compact automata to uncover this sequential structure automatically. We synthesise a human-interpretable automaton from trace data generated through exploration of the environment by the deep RL agent. The state space of the environment is then enriched with the synthesised automaton so that generation of a control policy by deep RL is guided by the discovered structure encoded in the automaton. The proposed approach is able to cope both with high-dimensional low-level features, and unknown sparse non-Markovian rewards. We have evaluated DeepSynth’s performance via a set of experiments including the Atari game Montezuma’s Revenge. Compared to existing approaches, we obtain a reduction of two orders of magnitude in the number of iterations required for policy synthesis, and also a significant improvement in scalability.

1 Introduction

Reinforcement Learning (RL) is the key enabling technique for a variety of applications of artificial intelligence, including advanced robotics (Polydoros and Nalpantidis 2017), resource and traffic management (Mao et al. 2016; Sadigh et al. 2014), drone control (Abbeel et al. 2007), chemical engineering (Zhou et al. 2017), and gaming (Mnih et al. 2015). While RL is a very general architecture, many advances in the last decade have been achieved using specific instances of RL that employ a deep neural network to synthesise optimal policies. A deep RL algorithm, AlphaGo (Silver et al. 2016), played moves in the game of Go that were initially considered glitches by human experts, but secured victory against the world champion. Similarly, AlphaStar (Vinyals et al. 2019) was able to defeat the world’s best players at the real-time strategy game StarCraft II, and to reach top 0.2% in scoreboards with an “unimaginably unusual” playing style.

While Deep RL can autonomously solve many tasks in complex environments, tasks that feature extremely sparse, non-Markovian rewards or other long-term sequential structures are often difficult or impossible to solve by unaided RL. A well-known example is the Atari game Montezuma’s Revenge, in which DQN (Mnih et al. 2015) failed to score. Interestingly, Montezuma’s Revenge and many other hard problems often require learning to achieve, possibly in a specific sequence, a set of high-level objectives to obtain the reward. These objectives can often be identified with passing through designated and semantically distinguished states of the system. This insight can be a lever that enables us to obtain a manageable, high-level model of the system’s behaviour and its dynamics.

Contribution: In this paper we propose DeepSynth, a new algorithm that automatically infers unknown sequential dependencies of a reward on high-level objectives and exploits this to guide a deep RL agent when the reward signal is history-dependent and significantly delayed. We assume that these sequential dependencies have a regular nature, as in formal language theory (Gulwani 2012). The identification of sequential dependencies on high-level objectives is the key to breaking down a complex task into a sequence of many Markovian ones. In our work, we use automata expressed in terms of high-level objectives to orchestrate sequencing of low-level actions in deep RL and to guide the learning towards sparse rewards. Furthermore, the automata representation allows a human observer to easily interpret the deep RL solution in a high-level manner, and to gain more insight into the optimality of that solution.

At the heart of DeepSynth is a model-free deep RL algorithm that is synchronised in a closed-loop fashion with an automaton inference algorithm, enabling our method to learn a policy that discovers and follows high-level sparse-reward structures. The synchronization is achieved by a product construction, which creates a hybrid architecture for the deep RL. When dealing with raw image input we assume an off-the-shelf unsupervised image segmentation method can provide enough object candidates, e.g. (Liu et al. 2019). We evaluate the performance of DeepSynth on a selection of benchmarks with sequential and unknown high-level structures. These experiments show that DeepSynth is able to automatically discover and formalise unknown, sparse, and non-Markovian high-level reward structures to efficiently synthesise successful policies in various domains when other related approaches fail. DeepSynth represents a better integration of deep RL and
formal automata synthesis than previous approaches, making learning for non-Markovian rewards more scalable.

**Related Work:** This work takes a formal approach to tackle the sparse reward problem in RL. In the RL literature dependencies of rewards upon objectives are often tackled with options (Sutton and Barto 1998) or, in general, are hierarchically structured. Current approaches to hierarchical RL, very much depend on state representations and whether they are structured enough for a suitable reward signal to be effectively engineered manually. Hierarchical RL therefore often requires detailed supervision in the form of explicitly specified high-level actions or intermediate supervisory signals (Precup 2001; Kears and Singh 2002; Daniel et al. 2012; Kulkarni et al. 2016; Vezhnevets et al. 2016; Bacon et al. 2017). A key difference between our approach and hierarchical RL is that our method produces a modular, human-interpretable and succinct automaton to represent the sequence of tasks, as opposed to complex and comparatively sample-inefficient structures, e.g. RNNs.

The closest line of work to ours, which aims to avoid these requirements, are model-based (Fu and Topcu 2014; Sadigh et al. 2014; Fulton and Platzer 2018; Cai et al. 2020) or recent model-free approaches in RL that constrain the agent with a temporal logic property (Hasanbeig et al. 2018; Toro Icarte et al. 2019; De Giacomo et al. 2019, 2020; Hasanbeig et al. 2020), which learns feasible tasks first and then stitches them together to accomplish a complex task. The key difference here is that the method assumes the policy sketches, i.e. temporal instructions, to be available to the agent. There is also recent work on learning underlying non-Markovian objectives when an optimal policy or human demonstration is available (Hasanbeig et al. 2019b).

Further related work is policy sketching (Andreas et al. 2017), which learns feasible tasks first and then stitches them together to accomplish a complex task. The key difference here is that the method assumes the policy sketches, i.e. temporal instructions, to be available to the agent. There is also recent work on learning underlying non-Markovian objectives when an optimal policy or human demonstration is available (Hasanbeig et al. 2019b).

2 Background on Reinforcement Learning

We first consider a conventional RL setup, consisting of an agent interacting with an environment, which is modelled as an unknown general Markov Decision Process (MDP):

**Definition 2.1 (General MDP)** The tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, s_0, \mathcal{P}, \mathcal{R}, \mathcal{L})$ is a general MDP over a set of continuous states $\mathcal{S} = \mathbb{R}^n$, where $\mathcal{A}$ is a set of actions, and $s_0 \in \mathcal{S}$ is the initial state. $\mathcal{P} : \mathcal{B}(\mathbb{R}^n) \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is a Borel-measurable conditional transition kernel that assigns to any pair of state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$ a probability measure $\mathcal{P}^\omega(\cdot | s, a)$ on the Borel space $(\mathcal{S}^\omega, \mathcal{B}(\mathcal{S}^\omega))$ (Bertsekas and Shreve 2004). $\mathcal{L}$ is called the vocabulary set and is a finite set of atomic propositions for which there exists a labelling function $L : \mathcal{S} \to 2^\mathcal{L}$ that assigns to each state $s \in \mathcal{S}$ a set of atomic propositions $L(s) \subseteq 2^\mathcal{L}$.

**Definition 2.2 (Path)** In a general MDP $\mathcal{M}$, an infinite path $\rho$ starting at $s_0$ is a sequence of states $\rho = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$ such that every transition $s_i \xrightarrow{a_i} s_{i+1}$ is possible in $\mathcal{M}$, i.e. $s_{i+1}$ belongs to the smallest Borel set $B$ such that $\mathcal{P}(B | s_i, a_i) = 1$. A finite path $\rho_n = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$ is a prefix of an infinite path. The set of infinite paths is $(\mathcal{S} \times \mathcal{A})^\omega$ and the set of finite paths is $(\mathcal{S} \times \mathcal{A})^* \times \mathcal{S}$.

At each state $s \in \mathcal{S}$, an agent action is determined by a policy $\pi$, which is a mapping from states to a probability distribution over the actions, i.e. $\pi : \mathcal{S} \to \mathcal{P}(\mathcal{A})$. Further, a random variable $R(s, a) \sim \mathcal{T}([s, a] \in \mathcal{P}(\mathcal{R}))$ is defined over the MDP $\mathcal{M}$, to represent the Markovian reward obtained when action $a$ is taken in a given state $s$, where $\mathcal{P}(\mathcal{R})$ is the set of probability distributions on subsets of $\mathcal{R}$, and $\mathcal{T}$ is the reward distribution. Similarly, a non-Markovian reward $\hat{R} : (\mathcal{S} \times \mathcal{A})^* \times \mathcal{S} \to \mathcal{R}$ is a mapping from the set of finite paths to real numbers and one possible realisation of $R$ and $\hat{R}$ at time step $n$ is denoted by $r_n$ and $\hat{r}_n$ respectively.

Due to space limitations we present the formal background on RL in (Hasanbeig et al. 2019b) and we only introduce the notations in the following. The expected discounted return for a policy $\pi$ and state $s$ is denoted by $U^\pi(s)$ which is
maximised by the optimal policy \( \pi^* \). Similarly, at each state the optimal policy maximises the Q-function \( Q(s, a) \) over the set of actions. The Q-function can be parameterised using a parameter set \( \theta^Q \) and updated by minimising a loss \( \mathcal{L}(\theta^Q) \).

### 3 Background on Automata Synthesis

The automata synthesis algorithm uses trace sequences to construct a succinct automaton that represents the behaviour exemplified by them, based on synthesis from examples approach (Gulwani 2012; Jeppu et al. 2020). This method makes an algorithmic improvement to scale to long traces by means of a segmentation approach, thus achieving automata learning in close-to-polynomial runtime as per empirical evidence (Jeppu 2020).

The synthesis framework takes as input a trace sequence and generates an \( N \)-state automaton conforming to the trace input. Starting with \( N = 1 \), the algorithm systematically searches for the required automaton, and incrementing \( N \) by 1 each time the search fails. This ensures that the smallest automaton conforming to the input trace is generated. The framework additionally uses two tunable hyper-parameters, \( w \) and \( l \). The hyper-parameter \( w \) is used to tackle growing algorithm complexity for long trace input. The synthesis framework divides the trace into segments using a sliding window of size \( w \) and unique segments are used for further processing. More specifically, the framework looks for the presence of patterns in traces. Multiple occurrences of these patterns are processed only once, thus reducing the size of the input to the algorithm.

Automata generated using only positive trace samples tend to overgeneralise (Gold 1978). This is mitigated by performing a compliance check of the automaton against the trace input to eliminate any transition sequences of length \( l \) that are accepted by the generated automaton but do not appear in the trace. The hyper-parameter \( l \) therefore controls the degree of generalisation in the generated automaton. A higher value for \( l \) yields more exact representations of the trace. The correctness of the generated automaton is verified by checking if the automaton accepts the input trace. If the check fails, missing trace data is incrementally added to refine the generated model, until the check passes. Further details about tuning these hyper-parameters are given in the following section.

### 4 DeepSynth

We begin by introducing the first level of Montezuma’s Revenge as a running example (Bellemare et al. 2013). Unlike other Atari games where the primary goal is limited to avoiding obstacles or collecting items with no particular order, Montezuma’s Revenge requires the agent to perform a long, complex sequence of actions before receiving any reward. The agent must find a key and open either door in Fig. 1.a. To this end, the agent has to climb down the middle ladder, jump on the rope, climb down the ladder on the right and jump over a skull to reach the key. The reward given by the Atari emulator for collecting the key is 100 and the reward for opening one of the doors is another 300. Due to the sparsity of the rewards most of the existing deep RL algorithms either fail to learn a policy that can even reach the key, e.g. DQN (Mnih et al. 2015), or the learning process is computationally heavy and sample inefficient, e.g. FeUdal (Vezhnevets et al. 2017), and Go-Explore (Ecoffet et al. 2019).

To overcome this problem various methods have been proposed that mostly hinge on intrinsic motivation and object-driven guidance. Unsupervised object detection (or unsupervised semantic segmentation) from raw image input has seen substantial progress in recent years, and became comparable to its supervised counterpart (Liu et al. 2019; Ji, Henriques, and Vedaldi 2019; Hwang et al. 2019; Zheng and Yang 2020). In this work, we assume an off-the-shelf image segmentation algorithm can provide plausible object candidates, e.g. (Liu et al. 2019). The key to solving a complex task such as Montezuma’s Revenge are the semantic correlations between the objects in the scene. In comparison, when a human player tries to solve this game the semantic correlations, such as “keys open doors”, are at least partially known and the player’s behaviour is driven by exploiting these known correlations and exploring unknown objects. This has been a subject of study in psychology, where animals and humans seem to have general motivations (often referred to as intrinsic motivations) that push them to explore and manipulate their environment, encouraging curiosity and cognitive growth (Berlyne 1960; Csikszentmihalyi et al. 1990; Ryan and Deci 2000).

Owing to the intuitive and modular structure of the automaton, DeepSynth can embed known correlations (if there exists any) into the learning algorithm for exploitation and pushes the exploration to infer the unknown parts. Such an automaton-based exploration scheme imitates biological cognitive growth in a formal and explainable way, and is driven by an intrinsic motivation to explore as many objects as possible in order to find the extrinsic optimal sequence of high-level Markovian objectives. To evaluate the full potential of DeepSynth, in all the experiments and examples of this paper we assume that semantic correlations are unknown to the agent. The agent starts with neither knowledge of the sparse reward task nor the correlation of the high-level objects.

Let us denote \( \Sigma \) as the set of detected objects. Note that the semantic meaning of the names for individual objects is not of relevance to the algorithm and \( \Sigma \)
In a general MDP $\mathcal{M}$, and over a
The learned automaton follows the standard definition of a Deterministic Finite Automaton (DFA) with the alphabet \( \Sigma_\mathfrak{A} \), where a symbol of the alphabet \( v \in \Sigma_\mathfrak{A} \) is given in the following by the labelling function \( L : \mathcal{S} \to 2^{\Sigma} \) defined earlier. Thus, given a trace sequence \( \sigma = \{v_i\}_{i=1}^n \) over a finite path \( \rho_n = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n \) in the MDP, the symbol \( v_i \in \Sigma_\mathfrak{A} \) is given by \( v_i = L(s_i) \).

The Atari emulator provides the number of lives left in the game, which is used to mark the reset of \( \Sigma_n \subseteq \Sigma_\mathfrak{A} \), where \( \Sigma_n \) is the set of labels the agent observed so far over a single life time. Upon losing a life, \( \Sigma_n \) is reset to the empty set.

**Definition 4.2 (Deterministic Finite Automaton)** A DFA \( \mathfrak{A} = (Q, q_0, \Sigma, F, \delta) \) is a state machine, where \( Q \) is a finite set of states, \( q_0 \in Q \) is the initial state, \( \Sigma \) is the alphabet, \( F \subseteq Q \) is the set of accepting states, and \( \delta : Q \times \Sigma \to Q \) is a transition function.

Let \( \Sigma_\mathfrak{A}^* \) be the set of all finite words over \( \Sigma_\mathfrak{A} \). A finite word \( w = v_1, v_2, \ldots, v_m \in \Sigma_\mathfrak{A}^* \) is accepted by a DFA \( \mathfrak{A} \) if there exists a finite run \( \theta \in \Omega^* \) starting from \( \theta_0 = q_0 \), where \( \theta_{i+1} = \delta(\theta_i, v_{i+1}) \) for \( i \geq 0 \) and \( \theta_m \in F \). Given the set of collected traces \( \mathcal{T} \) we construct a DFA using the approach described in Section 3. The algorithm first divides the trace into segments using a sliding window of size equal to the hyper-parameter \( w \) introduced earlier. The hyper-parameter \( w \) determines the input size, and consequently the algorithm runtime. Note that choosing \( w = 1 \) will not capture any sequential behaviour. For automata synthesis in DeepSynth, we would like to choose a value for \( w \) that results in the smallest input size, and thus we incrementally tried different values for \( w \), ranging within \( 1 < w \leq |\sigma| \), and have obtained the same automaton in all scenarios.

As discussed in Section 3, the hyper-parameter \( l \) controls the degree of generalisation in the learnt automaton. Learning exact automata from trace data is known to be NP-complete (Gold 1978). Thus, a higher value for \( l \) increases the algorithm runtime. We optimise over the hyper-parameters and choose \( w = 3 \) and \( l = 2 \) as the best fit for our setting. This ensures that the automata synthesis problem is not too complex for the synthesis framework to solve but at the same time it does not over-generalise to fit the trace.

The obtained automaton provides deep insight into the correlation of detected objects in Step 1 and shapes the intrinsic reward. The output of this stage is a DFA, from the set of succinct DFAs obtained earlier. Fig. 4 gives three exemplars of the evolution of the synthesised automaton for the segmented objects in Montezuma’s Revenge. Most of the deep RL approaches are able to reach to the stage of the DFA in Fig. 4.a via random exploration. However, reaching the key and further the doors as in Fig. 4.b and Fig. 4.c is challenging and is achieved by a hierarchical curiosity-driven learning method described in the next section. The automata synthesis is the “Synth” box in Fig. 2 and implementation details can be found in (Hasanbeig et al. 2019b).

**Deep Temporal Neural Fitted RL (Step 3 in Fig. 2):** We propose a deep RL scheme inspired by DQN (Mnih et al. 2015) (and Neural Fitted Q-iteration (NFQ) (Riedmiller 2005) when the input is in vector form, not raw image). We show that the proposed scheme (1) is able to synthesise a
policy whose traces are accepted by the DFA, (2) encourages the agent to explore under the DFA guidance, and (3) more importantly to expand the DFA towards task satisfaction.

Given the constructed DFA at each time step during the learning episode, if a new label is observed during exploration the intrinsic reward in (1) becomes positive. Namely,

$$R^i(s, a) = \begin{cases} \eta & \text{if } L(s') \notin \Sigma_a, \\ 0 & \text{otherwise} \end{cases}$$

where $\eta$ is an arbitrarily finite and positive reward, and $\Sigma_a$, as discussed in the Synthesis step, is the set of labels that the agent has observed in the current learning episode. Further, once a new label that does not belong to $\Sigma_0$ is observed during exploration (Step 1) it is then passed to the automation synthesis module (Step 2). The synthesis module then synthesises a new DFA that complies with the new label.

**Theorem 4.1 (Formalisation of The Intrinsic Reward)**
The optimal policies are invariant under the reward transformation in (1) and (2).

The proof of Theorem 4.1 is presented in (Hasanbeig et al. 2019b). In the following in order to explain the core ideas underpinning the algorithm, we temporarily assume that the MDP graph and the associated transition probabilities are fully known. Later we relax these assumptions, and we stress that the algorithm can be run model-free over any black-box MDP environment. We relate the black-box MDP and the automaton by synchronizing them on-the-fly (Remark 4.1) to create a new structure that breaks down a non-Markovian task into a set of Markovian, history-independent sub-goals.

**Definition 4.3 (Product MDP)** Given an MDP $M = (S, A, s_0, P, \Sigma)$ and a DFA $\mathcal{A} = (Q, q_0, \Sigma_0, F, \delta)$, the product MDP is defined as $(M \otimes \mathcal{A}) = M_\mathcal{A} = (S^\otimes, A, s_0^\otimes, P^\otimes, \Sigma^\otimes, F^\otimes)$, where $S^\otimes = S \times Q$, $s_0^\otimes = (s_0, q_0)$, $\Sigma^\otimes = Q$, and $F^\otimes = S \times F$. The transition kernel $P^\otimes$ is such that given the current state $(s_i, q_i)$ and action $a$, the new state $(s_j, q_j)$ is given by $s_j \sim P_i(a \mid s_i, q_i)$. The size of the replay buffer for each module is limited and in the case of our running example $|E_{q_i}| = 15000$. This includes the most recent frames that are observed when the product MDP state $s^\otimes = (s_i, q_i)$. In the running example we used RMsProp for each module with uniformly sampled minibatches of size 32. When the state is in vector form and no convolutional layer is involved we resort to NFQ deep RL modules instead of DQN modules for sample efficiency. The values of all hyperparameters are given in the extended version (Hasanbeig et al. 2019b).

**5 Experimental Results**

**Benchmarks and Setup:** We evaluate and compare the performance of DeepSynth with DQN on a comprehensive set of benchmarks, given in Table 1. The Minecraft environment (minecraft-tX) taken from (Andreas et al. 2017) requires solving challenging low-level control tasks, and features highly sequential high-level goals. The two mars-rover benchmarks are taken from (Hasanbeig et al. 2019d), and the models have uncountably infinite (continuous) state spaces. The example robot-surve is adopted from (Sadigh et al. 2014), and the task is to visit two regions in sequence while avoiding an unsafe area. Models s1p-easy and s1p-hard are inspired by the noisy MDPs of Chapter 6 in (Sutton and Barto 1998). The goal in s1p-easy is to reach a particular region of the MDP and the goal in s1p-hard is to visit four distinct regions sequentially in proper order. The frozen-lake
The task DFA column in Table 1 gives the number of states in the automaton that can be generated from the high-level objective sequences of the ground-truth task. The synth DFA column gives the number of states of the automaton synthesised by DeepSynth, and prod. MDP gives the number of states in the resulting product MDP (Definition 4.3). Finally, max sat. prob. at $s_0$ is the maximum probability of achieving the extrinsic reward from the initial state.

In all experiments the high-level objective sequences are initially unknown to the agent. Furthermore, the extrinsic reward is completing the task and reaching the objectives in the correct order. The details of all experiments are given in (Hasanbeig et al. 2019b, 2020a).

Results: The training progress for Montezuma’s Revenge and Task 3 in Minecraft is plotted in Fig. 5 and Fig. 6. In Fig. 6.a the orange line shows the very red deep net associated to the initial state of the DFA, the red and blue ones are of the intermediate states in the DFA and the green line is associated to the final state. This shows an efficient back-propagation of extrinsic reward from the final high-level state to the initial state, namely once the last deep net converges the expected reward is back-propagated to the second and so on. Each NFQ module has 2 hidden layers and 128 ReLUs.

Note that there may be a number of ways to accomplish a particular task in the synthesised DFAs. This, however, causes no harm since when the extrinsic reward is back-propagated, the non-optimal options fall out. Further, emphasise that if the task DFA, in any of the experiments, is known or even partially known a priori, then the Synth step can exploit this partial automaton by incrementally adding any new labels or subtask sequences discovered during exploration. As an example, if the automaton in Fig. 4.b is present initially, the agent is able to utilise the semantic correlation of the objects to facilitate its explorations and find the key faster.

### Table 1: Comparison between DeepSynth and DQN

<table>
<thead>
<tr>
<th>experiment</th>
<th>[S]</th>
<th>task DFA</th>
<th>synth DFA</th>
<th>prod. MDP</th>
<th>max sat. prob. at $s_0$</th>
<th>DeepSynth con. ep.</th>
<th>DQN con. ep.</th>
</tr>
</thead>
<tbody>
<tr>
<td>minecraft-t1</td>
<td>100</td>
<td>3</td>
<td>6</td>
<td>600</td>
<td>1</td>
<td>25</td>
<td>40</td>
</tr>
<tr>
<td>minecraft-t2</td>
<td>100</td>
<td>3</td>
<td>6</td>
<td>600</td>
<td>1</td>
<td>30</td>
<td>45</td>
</tr>
<tr>
<td>minecraft-t3</td>
<td>100</td>
<td>5</td>
<td>5</td>
<td>500</td>
<td>1</td>
<td>40</td>
<td>50</td>
</tr>
<tr>
<td>minecraft-t4</td>
<td>100</td>
<td>3</td>
<td>3</td>
<td>300</td>
<td>1</td>
<td>30</td>
<td>50</td>
</tr>
<tr>
<td>minecraft-t5</td>
<td>100</td>
<td>3</td>
<td>6</td>
<td>600</td>
<td>1</td>
<td>20</td>
<td>35</td>
</tr>
<tr>
<td>minecraft-t6</td>
<td>100</td>
<td>4</td>
<td>5</td>
<td>500</td>
<td>1</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>minecraft-t7</td>
<td>100</td>
<td>5</td>
<td>7</td>
<td>800</td>
<td>1</td>
<td>70</td>
<td>70</td>
</tr>
<tr>
<td>mars-rover-1</td>
<td>∞</td>
<td>3</td>
<td>∞</td>
<td>n/a</td>
<td>4</td>
<td>40</td>
<td>50</td>
</tr>
<tr>
<td>mars-rover-2</td>
<td>∞</td>
<td>4</td>
<td>∞</td>
<td>n/a</td>
<td>4</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>robot-survey</td>
<td>25</td>
<td>3</td>
<td>3</td>
<td>75</td>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>slip-easy-sel</td>
<td>120</td>
<td>2</td>
<td>2</td>
<td>240</td>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>slip-easy-med</td>
<td>400</td>
<td>2</td>
<td>2</td>
<td>800</td>
<td>1</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>slip-easy-rg</td>
<td>1000</td>
<td>2</td>
<td>2</td>
<td>3200</td>
<td>1</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>slip-hard-sel</td>
<td>120</td>
<td>5</td>
<td>5</td>
<td>720</td>
<td>1</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>slip-hard-med</td>
<td>400</td>
<td>5</td>
<td>5</td>
<td>2400</td>
<td>1</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>slip-hard-rg</td>
<td>1000</td>
<td>5</td>
<td>5</td>
<td>9600</td>
<td>1</td>
<td>120</td>
<td>120</td>
</tr>
<tr>
<td>frozen-lake-1</td>
<td>120</td>
<td>3</td>
<td>3</td>
<td>360</td>
<td>0.9983</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>frozen-lake-2</td>
<td>400</td>
<td>3</td>
<td>3</td>
<td>1200</td>
<td>0.9982</td>
<td>150</td>
<td>150</td>
</tr>
<tr>
<td>frozen-lake-3</td>
<td>1000</td>
<td>3</td>
<td>3</td>
<td>4800</td>
<td>0.9720</td>
<td>150</td>
<td>150</td>
</tr>
<tr>
<td>frozen-lake-4</td>
<td>120</td>
<td>6</td>
<td>6</td>
<td>840</td>
<td>0.9728</td>
<td>300</td>
<td>300</td>
</tr>
<tr>
<td>frozen-lake-5</td>
<td>400</td>
<td>6</td>
<td>6</td>
<td>2800</td>
<td>0.9722</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>frozen-lake-6</td>
<td>1000</td>
<td>6</td>
<td>6</td>
<td>11200</td>
<td>0.9467</td>
<td>450</td>
<td>450</td>
</tr>
</tbody>
</table>

* average number of episodes to convergence over 10 runs

Figure 5: Average episode reward progress in Montezuma’s Revenge with h-DQN (Kulkarni et al. 2016), DQN (Mnih et al. 2015), FeUdal-LSTM (Vezhnevets et al. 2017), Option-Critic (OC) (Bacon et al. 2017), inferring reward models (RM) (Icarte et al. 2019; Rens et al. 2020; Gaon and Brafman 2020; Xu et al. 2020) and DeepSynth with no intrinsic reward (DeepSynth-NI). h-DQN (Kulkarni et al. 2016) finds the door but only after 2M steps. FeUdal and LSTM find the door after 100M and 200M steps respectively. DQN, OC, RM and DeepSynth-NI remain flat.

Figure 6: Minecraft Task 3 Experiment: (a) Training progress with four hybrid deep NFQ modules, (b) Training progress with DeepSynth and DQN on the same training set $E$.

6 Conclusions

We have proposed a fully-unsupervised approach for training deep RL agents when the reward is extremely sparse and non-Markovian. We automatically infer this high-level structure from observed exploration traces using automata synthesis. The inferred automaton is a formal, un-grounded, human-interpretable representation of a complex task and its steps. Owing to the modular structure of the automaton, the overall task can be segmented into easy Markovian sub-tasks. Therefore, any segment of the proposed network that is associated with a sub-task can be used as a separate trained module in transfer learning. Another major contribution of the proposed method is that in problems where domain knowledge is available, it can be easily encoded as an automaton to guide learning. This enables the agent to solve complex tasks and saves the agent from an exhaustive exploration in the beginning. We showed that we are able to efficiently learn policies that achieve complex high-level objectives using fewer training samples as compared to alternative algorithms.
Acknowledgements
The authors would like to thank Hadrien Pouget for interesting discussions and the anonymous reviewers for their insightful suggestions. This work was supported by a grant from the UK NCSC, and Balliol College, Jason Hu scholarship.

References


Buzhinsky, I.; and Vyatkin, V. 2017. Modular plant model synthesis from behavior traces and temporal properties. ETFA 1–7.


Gulwani, S. 2012. Synthesis from Examples. In WAMBE.


